Hibernate

Persistencia usando Hibernate

Correspondencia entre el modelo relacional y el modelo de objetos

Las aplicaciones web, tales como un carrito de la compra online, van a utilizar bases de datos para almacenar sus datos de forma persistente.

Se entiende por persistencia la capacidad que tienen los objetos de conservar su estado e identidad entre distintas ejecuciones del programa que los creó o de otros programas que accedan a ellos. Las bases de datos relacionales son la opción más popular para almacenar datos.

La programación orientada a objetos y las bases de datos relacionales, se apoyan en dos paradigmas diferentes. El modelo relacional trata con relaciones, y conjuntos. Sin embargo, el paradigma orientado a objetos trata con objetos, sus atributos y asociaciones de unos con otros.

Hay una desaveniencia entre estos dos paradigmas, la también llamada diferencia objeto-relacional. Un mapeador objeto-relacional (u ORM para abreviar) nos ayudará evitar esta diferencia.

La diferencia objeto-relacional se amplia muy rápidametne si tienes grandes modelos de objetos. Y hay muchas más cosas a considerar como la carga lenta, las referencias circulares, el caché, etc. De hecho, se han realizado estudios que demuestran que el 35% del código de una aplicación se produce para mapear datos entre la aplicación y la base de datos.

Un ORM intenta reducir la mayoría de esa carga de trabajo. Con un buen ORM, sólo tienes que definir una vez la forma en que tus clases se mapean a tablas.

Desarrolladores Java acostumbrados a trabajar con JDBC probablemente se pregunten cuáles son las razones para utilizar un ORM.

Introducción a Hibernate

Introducción

Hibernate es una herramienta ORM completa que ha conseguido en un tiempo record una excelente reputación en la comunidad de desarrollo posicionándose claramente como el producto OpenSource líder en este campo gracias a sus prestaciones, buena documentación y estabilidad. Es valorado por muchos incluso como solución superior a productos comerciales dentro de su enfoque, siendo una muestra clara de su reputación y soporte la reciente integración dentro del grupo JBoss que seguramente generará iniciativas muy interesantes para el uso de Hibernate dentro de este servidor de aplicaciones.

Se empezó a desarrollar hace algo unos años por Gavin King siendo hoy Gavin y Christian Bauer los principales gestores de su desarrollo.

Hibernate parte de una filosofía de mapear objetos Java "normales", también conocidos en la comunidad como "POJOs" (Plain Old Java Objects), no contempla la posibilidad de automatizar directamente la persistencia de Entity Beans tipo BMP (es decir, generar automáticamente este tipo de objetos), aunque aún asi es posible combinar Hibernate con este tipo de beans utilizando los conocidos patrones para la delegación de persistencia en POJOs.

Una característica de la filosofía de diseño de Hibernate ha de ser destacada especialmente, dada su gran importancia: puede utilizar los objetos Java definidos por el usuario tal cual, es decir, no utiliza técnicas como generación de código a partir de descriptores del modelos de datos o manipulación de bytecodes en tiempo de compilación (técnica conocida por su amplio uso en JDO), ni obliga a implementar

interfaces determinados, ni heredar de una superclase. Utiliza en vez de ello el mecanismo de **reflexión** de Java.

Las razones que hacen que el uso de Hibernate sea muy importante son:

- ✓ Simplicidad y flexibilidad: necesita un único fichero de configuración en tiempo de ejecución y un documento de mapeo para cada aplicación. Este fichero puede ser el estándar de Java (extensión properties) o un fichero XML. También se tiene la alternativa de realizar la configuración de forma programática. El uso de frameworks de persistencia, tales como EJBs hace que la aplicación dependa del framework. Hibernate no crea esa dependencia adicional. Los objetos persistentes en la aplicación no tienen que heredar de una clase de Hibernate u obedecer a una semántica específica. Tampoco necesita un contenedor para funcionar.
- ✓ Completo: ofrece todas las características de orientación a objetos, incluyendo la herencia, tipos de usuario y las colecciones. Además, también proporciona una capa de abstracción SQL llamada HQL. Las sentencias HQL son compiladas por el framework de Hibernate y cacheadas para su posible reutilización.
- ✓ Prestaciones: uno de las grandes confusiones que aparecen al utilizar este tipo de frameworks es creer que las prestaciones de la aplicación se ven muy mermadas. Este no es el caso de Hibernate. La clave en este tipo de situaciones es si se realizan el número mínimo de consultas a la base de datos. Muchos frameworks de persistencia actualizan los datos de los objetos incluso cuando no ha cambiado su estado. Hibernate sólo lo hace si el estado de los objetos ha cambiado. El cacheado de objetos juega un papel importante en la mejora de las prestaciones de la aplicación. Hibernate acepta distintos productos de cacheado, tanto de código libre como comercial.

Conceptos básicos

Tipos de Objetos

Hibernate distingue entre dos tipos de objetos:

- ✓ Transient: sólo existen en memoria y no en un almacén de datos (recuérdese en este sentido también el modificador transient de Java), en algunos casos, no serán almacenados jamás en la base de datos y en otros es un estado en el que se encuentran hasta ser almacenados en ella. Los objetos transient han sido instanciados por el desarrollador sin haberlos almacenado mediante una sesión.
- ✓ Persistent: se caracterizan por haber sido ya almacenados y ser por tanto objetos persistentes. Han sido creados y almacenados en una sesión o bien devueltos en una consulta realizada con la sesión.

Sesión

Para almacenar y recuperar estos objetos de la base de datos, el desarrollador debe mantener una conversación con el motor de Hibernate mediante un objeto especial, la sesión (clase Session).

Sirve para delimitar una o varias operaciones relacionadas dentro de un proceso de negocio, demarcar una transacción y aporta algunos servicios adicionales como una caché de objetos para evitar interacciones innecesarias contra la Base de Datos.

Las sesiones son un concepto ligado a un proceso de negocio, por tanto es natural pensar que una sesión siempre va a pertener a un mismo thread de ejecución (el que pertenece a la ejecución de un método de negocio para un usuario o sistema externo concreto), aunque técnicamente se pueden compatir sesiones entre threads, esto no se debe hacer jamás por no ser una buena política de diseño y los consecuentes problemas que puede generar.

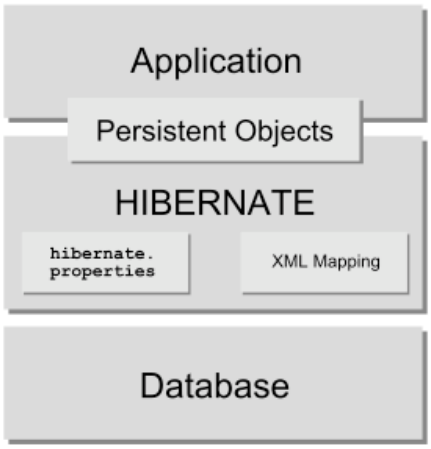
En un entorno multiusuario y por tanto multithread habrá por tanto múltiples sesiones simultáneas, cada una perteneciente a su correspondientes threads y con su contexto de objetos en caché, transacciones, etc.

Como tal no sorprende que las sesiones no son "thread-safe" y que la información vinculada a ella no sea visible para otras sesiones. Es también lógico que tenga que existir una "institución" superior para crear sesiones y realizar operaciones comunes a los diferentes threads como lo puede ser la gestión de una caché compartida entre threads o caché de segundo nivel.

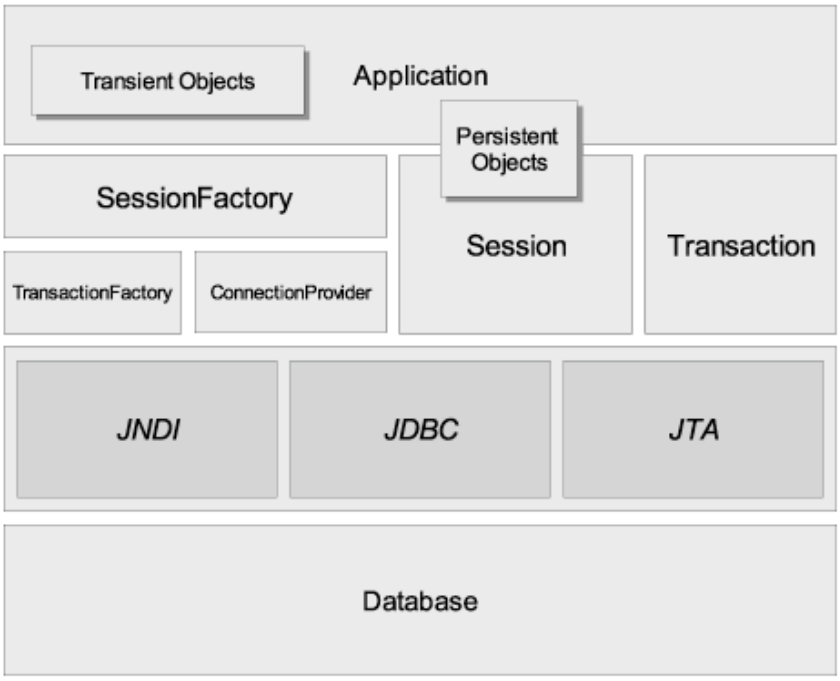
Este elemento es la clase SessionFactory y en ella podremos encontrar métodos como openSession() o evict(Class persistentClass).

Arquitectura Hibernate

Se analiza la arquitectura de Hibernate desde un punto de vista muy general:



Hibernate utiliza la base de datos y los ficheros de configuración para proporcionar servicios de persistencia a la aplicación.


Transient Objects
Application
Persistent Objects
SessionFactory
Session
Transaction
TransactionFactory
ConnectionProvider
JNDI
JDBC
JTA
Database

Entorno de desarrollo

Introducción

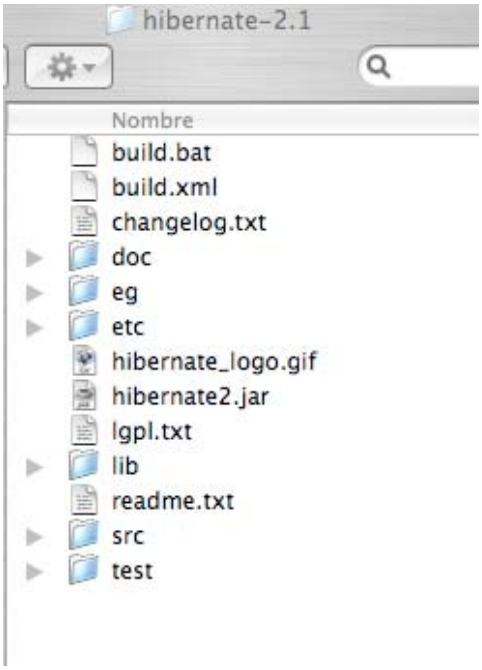
El principal objetivo en esta capítulo es conocer cuáles son los pasos a llevar a cabo para poder construir un proyecto con Hibernate. Para ello se deben realizar una serie de tareas generales:

- ✓ Obtener una distribución de Hibernate: el primer paso para crear un proyecto con Hibernate será obtener la última copia de Hibernate de su sitio web.
- ✓ Obtener Ant: (Java build tool) no es estrictamente necesaria pero no hará la vida más fácil.
- ✓ Configurar una base de datos.

Distribución de Hibernate

Bibliotecas	Descripción	Ficheros JAR/Zip
Hibernate	Clases de la API de Hibernate incluyendo la clase SchemaExportTask	hibernate2.jar
Hibernate-Extensions	Las clases que forman la extensión de Hibernate incluyendo la clase `Hbm2JavaTask`.	hibernate-tools.jar
Hibernate-lib	Clases de Hibernate auxiliares	dom4j-1.4.jar commons-logging-1.0.4.jar commons-collections-2.1.1.jar ehcache-0.9.jar cglib-full-2.0.2.jar jta.jar

Generalmente lo que se hará es incluir en nuestros proyectos la siguiente carpeta.



Ant

Ant es una herramienta del tipo de make (gnumake, nmake...) código abierto. Fué desarrollada dentro del Proyecto Apache Jakarta(http://jakarta.apache.org/ant) en Java.

Utiliza ficheros de configuración XML.

Para ejecutar Ant en línea de comandos se utiliza el comando ant:

- ✓ Por defecto busca el fichero build.xml en el directorio actual.
- ✓ -find busca build.xml en el directorio padre y siguientes hasta el raíz.
- ✓ -buildfile filename para indicar otro fichero.
- ✓ Se pueden especificar uno o más targetsa ejecutar.
  - × Por defecto ejecuta el targetindicado en el atributo default de la etiqueta <project>.
  - × Ej: ant –buildfile test.xml compile
- ✓ Se pueden establecer propiedades que sobreescribana las especificadas en el buildfile.
  - × -Dproperty=value
- ✓ Variables de entorno:ant-DMYVAR=$MYVAR ...
- ✓ Otras opciones:
  - × -help, -version, -quiet, -verbose, -projecthelp, -listener...

Los buildfiles son ficheros con formato xml que almacenan información de un determinado proyecto. Se diseñan para realizar una serie de acciones sobre el proyecto (targets).

- ✓ Etiqueta <project>:
  - × <name>: nombre del proyecto.
  - × <default>: target por defecto
  - × <basedir>: directorio base.
- ✓ Etiqueta <target>: es un objetivo compuesto por varias tareas que se ejecutan secuencialmente cuando se ejecuta el target.
  - × <name>: nombre del target, obligatorio
  - × <depends>: lista de targets de los cuales depende.

- <if> / <unless>
- <description>

Cada tarea a realizar en Ant está implementada en una clase Java:

Hay un conjunto de tareas incluidas en Antt, otras en .jar opcionales y, además, es posible incluir tareas propias. Algunas de esas tareas son:

- ✓ Ant: Llama a una target de otro buildfile.
- ✓ Copy: Copia ficheros y directorios.
- ✓ Delete: Borra ficheros y directorios.
- ✓ Echo: Envía un mensaje a System.out o a fichero.
- ✓ GZip: Crea un fichero Gzip.
- ✓ Javac: Compila código fuente Java.
- ✓ Javadoc: Genera ficheros HTML javadoc.
- ✓ Mkdir: Crea un directorio y todos los directorios padre necesarios.
- ✓ Property: Permite establecer valores de propiedades.
- ✓ Tar: Crea un fichero TAR a partir de un conjunto de ficheros.

Se muestra a continuación un ejemplo de fichero build.xml:

```
<?xml version="1.0"?>
<project name="HibernateJSF" default="schemaGenerator">

  <!--
      Definición de propiedades de sistema
  -->
  <property name="dirFuente" value="src"/>
  <property name="dirEjecutable" value="classes"/>
  <property name="hibernate" value="hibernate-2.1"/>
```

```
<property name="hibernate.mappings" value="mappings"/>
<!--<property name="jdbc" value="C:\oracle\product\10.1.0\Db_1\jdbc"/> -->
<property name="hibernate.extensions" value="tools"/>
<property name="hibernate.properties" value="properties"/>

<!--
      Rutas que se van a utilizar dentro de nuestras tareas

-->
<path id="project.class.path">
<pathelement location="${dirEjecutable}" />

  <!-- Rutas asociadas a las librerias de Hibernate -->
    <!-- 1ª ruta: /hibernate-2.1/hibernate2.jar -->
    <fileset dir="${hibernate}">
      <include name="hibernate2.jar"/>
    </fileset>
    <!-- 2ª ruta: /hibernate-2.1/lib/*.jar -->
    <fileset dir="${hibernate}/lib">
      <include name="*.jar"/>
    </fileset>
    <!-- 3ª ruta: /hibernate-2.1/tools/*.jar -->
    <fileset dir="${hibernate.extensions}/lib">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${hibernate.extensions}">
      <include name="hibernate-tools.jar"/>
    </fileset>

  <!-- ************************************************** -->
    <fileset dir=".">
      <include name="mysql-connector-java-3.0.17-ga-bin.jar"/>
    </fileset>
</path>
```

```
<!--
     Se crean dos directorios, primero para el codigo de la
     clase que va a ser creada automaticamente y el segundo para
     los ficheros .class
-->
<target name="init">
  <mkdir dir="${dirFuente}"/>
  <mkdir dir="${dirEjecutable}"/>
</target>


<!-- ************************************************************
     Se define una tarea que genera automaticamente una
     clase Java a partir de información de configuración:
     Catalog.hbm.xml
-->
<!-- ************************************************************ -->
<taskdef name="javaGen"
         classname="net.sf.hibernate.tool.hbm2java.Hbm2JavaTask"
         classpathref="project.class.path"
/>
<target name="javaGenerator" depends="init">
  <javaGen output="${dirFuente}">
    <fileset dir="${hibernate.mappings}">
      <include name="Catalog.hbm.xml"/>
    </fileset>
  </javaGen>
</target>
<!-- ************************************************************ -->

<!-- ************************************************************
     Se define una tarea para compilar las clases que han sido generadas
```

```
      automáticamente
  -->
  <target name="compile" depends="javaGenerator">
     <javac srcdir="${dirFuente}"
         destdir="${dirEjecutable}">
         <classpath refid="project.class.path"/></javac>
  </target>

  <!-- ********************************************************** -->

  <taskdef name="schemaGen"
           classname="net.sf.hibernate.tool.hbm2ddl.SchemaExportTask"
           classpathref="project.class.path"/>

  <target name="schemaGenerator" depends="compile">
    <schemaGen properties="${hibernate.properties}/hibernate.properties"
quiet="no">
        <fileset dir="${hibernate.mappings}">
            <include name="Catalog.hbm.xml"/>
        </fileset>
    </schemaGen>
  </target>

</project>
```

Base de Datos

Con respecto a la base de datos, sólo decir que el driver JDBC para ella debe incluirse en nuestro proyecto.

Configuración

Introducción

Debido a que Hibernate ha sido diseñado para que pueda trabajar en distintos entornos, existen una gran cantidad de parámetros de configuración.

Hibernate es un sistema altamente configurable para adaptarse a esto.

Para configurar Hibernate se deben realizar:

- ✓ Configuración del servicio Hibernate.
- ✓ Proporcionar a Hibernate toda la información asociada a las clases que se quieren hacer persistentes. La configuración de estas clases permitirá eliminar ese salto entre objetos y bases de datos relacionales.

Configuración básica

Hibernate proporciona dos posibles ficheros de configuración:

- ✓ Fichero de propiedades estándar de Java (hibernate.properties).
- ✓ Fichero XML normalmente llamado hibernate.cfg.xml.

Ambos ficheros llevan a cabo la misma tarea (configuración del servicio Hibernate). Si ambos ficheros se encuentran en el classpath de la aplicación, el segundo sobreescribe al primero.

Se muestra a continuación un ejemplo del fichero hibernate.cfg.xml:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-
```

```
3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.username">uid</property>
    <property name="connection.password">pwd</property>
    <property name="connection.url">
        jdbc:mysql://localhost/db
    </property>
    <property name="connection.driver_class">
        com.mysql.jdbc.Driver
</property>
    <property name="dialect">
        org.hibernate.dialect.MySQLDialect
    </property>
    <mapping resource="com/manning/hq/ch03/Event.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Location.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Speaker.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Attendee.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Aunque es bastante habitual encontrar una configuración algo distinta combinando ambos ficheros:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="show_sql">true</property>
        <mapping resource="com/manning/hq/ch03/Event.hbm.xml"/>
        <mapping resource="com/manning/hq/ch03/Attendee.hbm.xml"/>
```

```
        <mapping resource="com/manning/hq/ch03/Speaker.hbm.xml"/>
        <mapping resource="com/manning/hq/ch03/Location.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Fichero de propiedades hibernate.properties

```
hibernate.connection.username=username
hibernate.connection.password=password
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/events_calendar
hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Estos ficheros se utilizan para configurar el tipo de conexión que se va a generar contra la base de datos y las clases que se van a asociar a las tablas.

Ficheros de mapeo

Son usados para proporcionar toda la información necesaria para asociar objetos a la tablas de la base de datos con la que queremos trabajar.

Las definiciones de mapeo pueden guardarse juntas en un único fichero de mapeo. Existe también la posibilidad de generar un fichero para cada uno de los objetos. Esta segunda opción es la más recomendable.

La extensión para este tipo de ficheros es .hbm.xml. Por ejemplo, si tenemos una clase llamada Usuario, su fichero de mapeo asociado será Usuario.hbm.xml.

```
<?xml version="1.0"?><!DOCTYPE hibernate-mapping PUBLIC "-
//Hibernate/Hibernate Mapping DTD 2.0//EN"
```

```
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
 <class name="entidades.Usuario" table="Usuario">
    <id name="id" type="int" column="UsuarioId">
      <generator class="native"/>
    </id>
    <property name="nombre" type="string"/>
    <property name="primerApellido" column="apellido1" type="string"/>
    <property name="segundoApellido" column="apellido2" type="string"/>
    <property name="direccion" type="string"/>
    <property name="telefono" type="int"/>
  </class>
</hibernate-mapping>
```

Inmediatamente después de la etiqueta <hibernate-mapping> se encuentra la etiqueta <class ... > que indica el inicio de la definición de mapeo para una específica clase.

El atributo table define el nombre de la tabla de la base de datos relacional usada para almacenar el estado de los objetos.

La etiqueta id describe la clave primaria para la clase persistente. Cada clase persistente debe tener un id declarando una clave primaria en la tabla. El atributo name define la propiedad de la clase que almacenará el valor de la clave primaria.

Cada etiqueta property corresponde con una propiedad en el objeto.

Conexión a la base de datos y generación de un esquema

Para llevar a cabo una conexión con una base de datos se debe incorporar una serie de datos en ficheros de configuración. Es bastante habitual distribuir esta configuración en los dos posibles ficheros que se pueden utilizar:

- ✓ Fichero hibernate.properties:

```
hibernate.connection.username=username
hibernate.connection.password=password
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/events_calendar
hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Como se muestra, se debe configurar la conexión incorporando en el fichero los datos habituales (nombre de usuario, contraseña, url, ...). La información de mapeo se incorporaría en el fichero xml:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="show_sql">true</property>
        <mapping resource="com/manning/hq/ch03/Event.hbm.xml"/>
        <mapping resource="com/manning/hq/ch03/Attendee.hbm.xml"/>
        <mapping resource="com/manning/hq/ch03/Speaker.hbm.xml"/>
        <mapping resource="com/manning/hq/ch03/Location.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

- ✓ Otra posible opción es incluir toda la información de configuración en el fichero xml hibernate.cfg.xml:

```
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.username">uid</property>
    <property name="connection.password">pwd</property>
    <property name="connection.url">
        jdbc:mysql://localhost/db
    </property>
    <property name="connection.driver_class">
        com.mysql.jdbc.Driver
</property>
    <property name="dialect">
        org.hibernate.dialect.MySQLDialect
    </property>
    <mapping resource="com/manning/hq/ch03/Event.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Location.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Speaker.hbm.xml"/>
    <mapping resource="com/manning/hq/ch03/Attendee.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Añadir tablas es una tarea muy habitual cuando se trabaja con bases de datos (CREATE TABLE).

Para realizar esta tarea Hibernate incluye una tarea Ant (SchemaExport) que, apoyándose en ficheros de mapeo, clases y el fichero de configuración hibernate.cfg.xml, genera tablas en una base de datos. El fichero build.xml estaría configurado de la siguiente manera:

```
...
<target name="schema-export" depends="compile" >
      <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask">
          <classpath refid="runtime.classpath" />
      </taskdef>
      <schemaexport config="${src.java.dir}/hibernate.cfg.xml"/>
 </target>
…
```

Esta tarea necesita conocer:

- ✓ Clases compiladas.
- ✓ Biblioteca Hibernate.
- ✓ Driver MySql.
- ✓ Fichero hibernate.cfg.xml.

Creación de clases persistentes

Las clases persistentes son clases que en una aplicación implementan las entidades del problema de negocio (por ejemplo: Cliente, Pedido, etc).

Hibernate trabaja con éstas siempre y cuando sean POJO's (Plain Old Java Object). Veamos un ejemplo:

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;
    private Cat mother;
    private Set kittens = new HashSet();
    private void setId(Long id) {
          this.id=id;
    }
    public Long getId() {
          return id;
    }
    void setBirthdate(Date date) {
          birthdate = date;
    }
    public Date getBirthdate() {
          return birthdate;
    }
```

```
    void setWeight(float weight) {
           this.weight = weight;
    }
    public float getWeight() {
           return weight;
    }
    public Color getColor() {
           return color;
    }
    void setColor(Color color) {
           this.color = color;
    }
    void setSex(char sex) {
           this.sex=sex;
    }
    public char getSex() {
           return sex;
    }
    void setLitterId(int id) {
           this.litterId = id;
    }
    public int getLitterId() {
           return litterId;
    }
    void setMother(Cat mother) {
           this.mother = mother;
    }
    public Cat getMother() {
           return mother;
    }
    void setKittens(Set kittens) {
           this.kittens = kittens;
    }
    public Set getKittens() {
```

```
            return kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
            kitten.setMother(this);
            kitten.setLitterId( kittens.size() );
            kittens.add(kitten);
    }
}
```

Estas clases deben cumplir cuatro condiciones:

- ✓ Implementar un constructor sin argumentos.
- ✓ Proporcionar una propiedad identificadora (opcional).
- ✓ Clase no final (opcional).
- ✓ Métodos getter y setter para cada campo (opcional).

Estas clases deben estar asociadas a un fichero de mapeo. Veamos un ejemplo sencillo. Supongamos que tenemos la clase Usuario con el siguiente código:

```
package hibernatejsf;

public class Usuario {

    private Integer id;
    private String nombre;
    private String primerApellido;
    private String segundoApellido;
    private String direccion;
    private Integer telefono;

    public Usuario() {
```

```
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public Integer getId() {
        return id;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getNombre() {
        return nombre;
    }
    public void setPrimerApellido(String primerApellido) {
        this.primerApellido = primerApellido;
    }
    public String getPrimerApellido() {
        return primerApellido;
    }
    public void setSegundoApellido(String segundoApellido) {
        this.segundoApellido = segundoApellido;
    }
    public String getSegundoApellido() {
        return segundoApellido;
    }
    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
    public String getDireccion() {
        return direccion;
    }
    public void setTelefono(Integer telefono) {
        this.telefono = telefono;
    }
```

```
    public Integer getTelefono() {
        return telefono;
    }
```

Su fichero de mapeo se muestra a continuación:

```
<?xml version="1.0"?><!DOCTYPE hibernate-mapping PUBLIC "-
//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
  <class name="entidades.Usuario" table="Usuario">
    <id name="id" type="int" column="UsuarioId">
      <generator class="native"/>
    </id>
    <property name="nombre" type="string"/>
    <property name="primerApellido" column="apellido1" type="string"/>
    <property name="segundoApellido" column="apellido2" type="string"/>
    <property name="direccion" type="string"/>
    <property name="telefono" type="int"/>
  </class>
</hibernate-mapping>
```

Hibernate proporciona la posibilidad de crear clases persistentes diseñando única y exclusivamente ficheros de mapero (mapping files). Esto es lo que se conoce como Modelo Dinámico. Para realizar esta tarea Hibernate incluye una tarea Ant:

```
...
```

```
  <taskdef name="javaGen"
           classname="net.sf.hibernate.tool.hbm2java.Hbm2JavaTask"
           classpathref="project.class.path"
  />
  <target name="javaGenerator" depends="init">
    <javaGen output="${dirFuente}">
      <fileset dir="${hibernate.mappings}">
        <include name="Catalog.hbm.xml"/>
      </fileset>
    </javaGen>
  </target>
...
```

Trabajando con colecciones

Las colecciones en Java forman parte del Java Foundation Classes (JFC) desde el lanzamiento de JDK 1.2. Su API desde un principio ha sido muy utiliizada por los desarrolladores gracias a su flexibilidad.

Hibernate proporciona soporte para las interfaces:

- ✓ java.util.List.
- ✓ java.util.Map.
- ✓ java.util.Set.

Hibernate ofrece distintas formas para almacenar colecciones.

El almacenamiento de Colecciones es muy sencillo.

Cuando Hibernate almacena una colección, retiene también toda la semántica asociada a su interface. Por ejemplo, cuando una lista (interface List) es almacenada, el índice que identifica el orden también es almacenado. De esa forma cuando ésta sea recuperada de la base de datos su comportamiento será idéntico al inicial. Esto es muy interesante, pero qué ocurre cuando se quiere almacenar una colección sin preocuparse de la semántica asociada.

Para este tipo de situaciones Hibernate tienen unos tipos llamados bag. Para evitar confusiones se muestra a continuación una tabla con un listado de todos los tipos:

Hibernate collection type	**Java collection type**	**Description**
set	java.util.Set	Persists an unordered, unique collection of values or objects.
map	java.util.Map	Persists a collection of key/value pairs.
list	java.util.List	Persists an ordered, non-unique collection of values or objects.

bag	java.util.List	Persists an unordered, non-unique collection of values or objects. array N/A Persists an indexed, non-unique collection of values or objects.
primitive-array	N/A	Persists an indexed, non-unique collection of primitive values.
idbag	java.util.List	Persists an unordered, non-unique, many-to-many collection using a surrogate key.

Veamos cómo se mapean colecciones mediante un ejemplo:

```
<hibernate-mapping package="com.manning.hq">
    <class name="Event" table="events">
        ...
        <set name="speakers">
            <key column="event_id"/>
            <one-to-many class="Speaker"/>
        </set>
        <set name="attendees">
            <key column="event_id"/>
            <one-to-many class="Attendee"/>
        </set>
        ...
    </class>
</hibernate-mapping>
```

Corresponde a la siguiente clase:

```
public class Event {
    private Set speakers;
    private Set attendees;
    public void setSpeakers(Set speakers) {
        this.speakers = speakers;
```

```
    }
    public Set getSpeakers() {
          return this.speakers;
}
    public void setAttendees(Set attendees) {
          this.attendees = attendees;
    }
    public Set getAttendees () {
          return this.attendees;
    }
    ...
}
```

Usando objetos persistentes

Configuración de SessionFactory

La interface SessionFactory proporciona instancias de objetos Session, que representan a la base de datos. Éstas son generalmente compartidas por toda la aplicación, en cambio, las instancias de sesión deben ser usadas sólo por una única transacción o unidad de trabajo.

Method Summary

void	**close**() Destroy this SessionFactory and release all resources (caches, connection pools, etc).
void	**evict**(Class persistentClass) Evict all entries from the second-level cache.
void	**evict**(Class persistentClass, Serializable id) Evict an entry from the second-level cache.
void	**evictCollection**(String roleName) Evict all entries from the second-level cache.
void	**evictCollection**(String roleName, Serializable id) Evict an entry from the second-level cache.
void	**evictQueries**() Evict any query result sets cached in the default query cache region.
void	**evictQueries**(String cacheRegion) Evict any query result sets cached in the named query cache region.
Map	**getAllClassMetadata**() Get all ClassMetadata as a Map from Class to metadata object
Map	**getAllCollectionMetadata**() Get all CollectionMetadata as a Map from role name to metadata object
ClassMetadata	**getClassMetadata**(Class persistentClass) Get the ClassMetadata associated with the given entity class
CollectionMetadata	**getCollectionMetadata**(String roleName) Get the CollectionMetadata associated with the named collection role
SQLExceptionConverter	**getSQLExceptionConverter**() Retrieves the SQLExceptionConverter in effect for this SessionFactory.
Databinder	**openDatabinder**() Create a new databinder.
Session	**openSession**() Create database connection and open a Session on it.
Session	**openSession**(Connection connection) Open a Session on the given connection.
Session	**openSession**(Connection connection, Interceptor interceptor) Open a Session on the given connection, specifying an interceptor.
Session	**openSession**(Interceptor interceptor) Create database connection and open a Session on it, specifying an interceptor.

Para configurar SessionFactory tendremos que apoyarnos en la clase Configuration. Es usada para cargar los ficheros de mapeo y crear la Sessionfactory. Existen tres formas distintas de crear e inicializar el objeto Configuration.

```
Configuration configuracion = new Configuration();
SessionFactory f= configuracion.configure().buildSessionFactory() ;
```

El método configure() hace que Hibernate cargue el fichero hibernate.cfg.xml. Sin esto el único fichero que sería cargado sería hibernate.properties. Además, la clase Configuration puede cargar ficheros de mapeo de forma programática:

```
Configuration configuracion = new Configuration();
configuracion.addFile("com/manning/hq/ch03/Event.hbm.xml");
SessionFactory f= configuracion.configure().buildSessionFactory() ;
```

Existe otra posibilidad de hacer lo mismo:

```
Configuration configuracion = new Configuration();
configuracion.addClass("com.manning.hq.ch03.Event.class");
SessionFactory f= configuracion.configure().buildSessionFactory() ;
```

Cuando el número de ficheros de mapeo crece demasiado es recomendable agruparlos todos en un fichero con extensión .jar.

Haciendo persistentes los objetos

Llevar a cabo esta tarea es tan simple como guardar dicho objeto en la instancia de la sesión:

Method Summary

Transaction	**beginTransaction**() Begin a unit of work and return the associated `Transaction` object.
void	**cancelQuery**() Cancel execution of the current query.
void	**clear**() Completely clear the session.
Connection	**close**() End the `Session` by disconnecting from the JDBC connection and cleaning up.
Connection	**connection**() Get the JDBC connection.
boolean	**contains**(Object object) Check if this instance is associated with this `Session`.
Criteria	**createCriteria**(Class persistentClass) Create a new `Criteria` instance, for the given entity class.
Query	**createFilter**(Object collection, String queryString) Create a new instance of `Query` for the given collection and filter string.
Query	**createQuery**(String queryString) Create a new instance of `Query` for the given query string.
Query	**createSQLQuery**(String sql, String[] returnAliases, Class[] returnClasses) Create a new instance of `Query` for the given SQL string.
Query	**createSQLQuery**(String sql, String returnAlias, Class returnClass) Create a new instance of `Query` for the given SQL string.
void	**delete**(Object object) Remove a persistent instance from the datastore.

. . .

```
// Crear un objeto
Usuario u= new Usuario();
u.setNombre("Antonio");
u.setPrimerApellido("Cruz");
...

Configuration configuracion = new Configuration();
configuracion.addClass("com.manning.hq.ch03.Usuario.class");
SessionFactory f= configuracion.configure().buildSessionFactory() ;
Session sesion= f.openSession();
sesion.save(u);
sesion.flush();

...
```

Si se quisiera actualizar se podría utilizar el método update.

Devolviendo objetos

El método load nos permite recuperar objetos siempre y cuando conozcamos el identificador de dicho objeto:

```
Event event = (Event) session.load(Event.class, eventId);
session.close();
```

El problema surge cuando no se conoce el identificador del objeto. Ese es el momento en el que se debe utilizar HQL.

Veamos varios ejemplos:

```
Query query = session.createQuery("from Event");
```

```
List events = query.list();
```

HQL (Hibernate Query Language)

Introducción

Las consultas en Hibernate se encuentran estructuradas de forma similar al tradicional SQL con las típicas claúsulas SELECT, FROM y WHERE.

Se muestra a continuación una consulta simple:

```
from Usuario
```

Esta consulta devuelve todas las instancias a Event que se encuentran en la base de datos.

Existen dos formas dentro de las APIs de Hibernate que nos permiten ejecutar consultas:

Métodos find

List	**find**(String query) Execute a query.
List	**find**(String query, Object[] values, Type[] types) Execute a query with bind parameters.
List	**find**(String query, Object value, Type type) Execute a query with bind parameters.

Por ejemplo:

```
List results = session.find("from Event");
```

La interface Session tiene un método iterador que nos permite recorrer la lista de forma cómoda:

```
Iterator results = session.iterate("from Event");
while (results.hasNext()) {
    Event myEvent = (Event) results.next();
    // ...
}
```

Interface Query

Method Summary	
String[]	**getNamedParameters**() Return the names of all named parameters of the query.
String	**getQueryString**() Get the query string.
Type[]	**getReturnTypes**() Return the Hibernate types of the query result set.
Iterator	**iterate**() Return the query results as an `Iterator`.
List	**list**() Return the query results as a `List`.
ScrollableResults	**scroll**() Return the query results as `ScrollableResults`.
ScrollableResults	**scroll**(ScrollMode scrollMode) Return the query results as `ScrollableResults`.
Query	**setBigDecimal**(int position, BigDecimal number)
Query	**setBigDecimal**(String name, BigDecimal number)
Query	**setBinary**(int position, byte[] val)
Query	**setBinary**(String name, byte[] val)
Query	**setBoolean**(int position, boolean val)

Esta interface es creada por la sesión y proporciona mayor control sobre los objetos devueltos:

```
Query q = session.createQuery("from Event");
List results = q.list();
```

Si se quiere limitar el número de resultados:

```
Query q = session.createQuery("from Event");
q.setMaxResults(15);
List results = q.list();
```

Query también proporciona un iterador para realizar recorridos sobre las colecciones de datos, método iterate().

En determinadas ocasiones podemos hacer que se comporte como un PreparedStatement de JDBC:

```
Query q = session.createQuery("from Event where name = ? ");
q.setParameter(0, "Opening Plenary");
List results = q.list();
```

No se necesita indicar el tipo del parámetro, pero es posible:

```
q.setParameter(0, "Opening Plenary", Hibernate.STRING);
```

Existe otro mecanismo muy interesante llamado named parameters (:nombreParametro). Se muestra un ejemplo ilustrativo:

```
Query q = session.createQuery("from Event where name = :name");
q.setParameter("name", "Opening Plenary");
List results = q.list();
```

Este parámetro puede aparecer en la consulta tantas veces como sea necesario:

```
Query q = session.createQuery("from Event where "+
    "startDate = :startDate or endDate < :startDate");
q.setParameter("startDate", eventStartDate);
List results = q.list();
```

Claúsulas HQL

Claúsula FROM

Esta claúsula te permite especificar los objetos que estás buscando. Permite crear alias. Ejemplo:

```
from Event e where e.name='Opening Plenary'
```

Nota: No está permitido el uso de as.

Claúsula JOIN

Se quieren devolve todos los eventos que han sido atendidos por el elemento identificado por el siguiente id:

```
from Event e join e.attendees a where a.id=314
```

Claúsula SELECT

Esta claúsula permite especificar una lista de valores a ser devueltos por la consulta. Esta claúsula no está forzada a devolver objetos completos, puede devolver campos aislados:

```
select e.name from Event e
```

Devuelve una lista de cadenas con el nombre de los eventos.

```
select e.name, e.startDate from Event e
```

Cada elemento en la lista devuelta es un array de Object con dos elementos.

Veamos un ejemplo algo mayor:

```
Session session = factory.openSession();
String query = " select e.name, e.startDate from Event e ";
Query query = session.createQuery("query");
List results = query.list();
for (Iterator I = results.iterator(); i.hasNext();) {
    Object[] row = (Object[]) i.next();
    String name = (String) row[0];
    Date startDate = (Date) row[1];
    // ...
```

Hibernate Caching

Cacheado de sesión

Una forma sencilla de mejorar las prestaciones de una aplicación dentro del servicio Hibernate, es el cacheo de objetos.

El almacenamiento de objetos en memoria evita a Hibernate tener que acceder a la base de datos cada vez que necesita devolver un objeto.

Se muestra un ejemplo ilustrativo:

```
Session session = factory.openSession();
Event e = (Event) session.load(Event.class, myEventId);
e.setName("New Event Name");
session.saveOrUpdate(e);
// later, with the same Session instance
Event e = (Event) session.load(Event.class, myEventId);
e.setDuration(180);
session.saveOrUpdate(e);
session.flush();
```

Se devuelve una instancia de un objeto, éste es cacheado por la sesión internamente. Todas las actualizaciones realizados sobre ese objeto son combinadas en una única actualización cuando se flushea la sesión.

La interface Session tiene acceso a una caché para cada objeto que es cargado y guardado durante el tiempo de vida de esa sesión.

Transacciones Hibernate

Introducción

Una transacción es un conjunto de operaciones agrupada en una única unidad de trabajo. Si alguna operación de ese conjunto falla todas las operaciones se deshacen. Hibernate puede ejecutarse en muy diferentes entornos soportando distintos tipos de transacciones. Aplicaciones independientes y algún servidor sólo soportan transacciones JDBC, en cambio, hay otras que soportan JTA (Java Transaction API).

Hibernate necesita una forma de abstraer las distintas estrategías en función del entorno en el que se encuentra. Se define la clase y la interface Transaction para afrontar ese problema:

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();
Event event = new Event();
// ... populate the Event instance
session.saveOrUpdate(event);
tx.commit();
```

Interface Transaction

Method Summary	
void	**commit**() Flush the associated `Session` and end the unit of work.
void	**rollback**() Force the underlying transaction to roll back.
boolean	**wasCommitted**() Check if this transaction was successfully committed.
boolean	**wasRolledBack**() Was this transaction rolled back or set to rollback only?

Clase Transaction

Esta clase implementa la interface Transaction y ofrece un conjunto de métodos:

Method Summary	
void	**abort**() Abort the transaction.
void	**begin**() Begin the transaction.
void	**checkpoint**() Commit the changes, but leave the transaction open.
void	**commit**() Commit the transaction.
Session	**getSession**() Get the underlying Hibernate `Session`.
boolean	**isOpen**() Is the transaction open.
void	**join**() Associate the current thread with this `Transaction`.
void	**leave**() Disassociate the thread the `Transaction`.
void	**lock**(Object obj, int lockMode) Obtain a lock upon the given object.
boolean	**tryLock**(Object obj, int lockMode) Not implemented.

Configuración de Transacciones

La propiedad transaction.factory_class define cuál es la estrategia de transacciones que utiliza Hibernate. Por defecto se utilizan transacciones JDBC. Para usar JTA se debe configurar el fichero hibernate.cfg.xml de la siguiente manera:

```
...
<property name="transaction.factory_class">
    org.hibernate.transaction.JTATransactionFactory
</property>
<property name="jta.UserTransaction">
    java:comp/UserTransaction
</property>
...
```

Hibernate y el patrón DAO

La mayoría de los programadores Java están familiarizados con el patrón Data Access Object (DAO). Es uno de los patrones del Core de Sun y suele ser mencionado habitualmente en muchos libros de Java. Es el patrón más importante para aplicaciones que almacén de datos persistente. Es habitualmente muy usado en aplicaciones que trabajan con SQL, por lo tanto el uso en Hibernate tiene bastante sentido.

El objetivo de este patrón es responder a una sencilla pregunta:

- ✓ Lugar donde localizar el código asociado al acceso a datos: la inexperiencia hace que el código de acceso se encuentre distribuido por la aplicación. Esto supone un problema cuando se quieren introducir modificaciones. Este patrón pretende mantener todo el código SQL junto. Si eso es bueno para SQL, también lo será para HQL.
- ✓ Si una aplicación almacena todo el código HQL en un único lugar será mucho más fácil de mantener y modificar.

Veamos una clase ejemplo:

```
package com.manning.hq.ch07;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.Query;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import java.util.List;
```

```
public class SimpleEventDao {

    Log log = LogFactory.getLog(SimpleEventDao.class);
    private Session session;
    private Transaction tx;

    public SimpleEventDao() {

        HibernateFactory.buildIfNeeded();

    }

    /**
     * Insert a new Event into the database.
     * @param event
     */

    public void create(Event event) throws DataAccessLayerException {
        try {
           startOperation();
           session.save(event);
           tx.commit();

        } catch (HibernateException e) {

            handleException(e);

        } finally {

            HibernateFactory.close(session);

        }

    }
```

```
/**
 * Delete a detached Event from the database.
 * @param event
 */

public void delete(Event event) throws DataAccessLayerException {

    try {

        startOperation();

        session.delete(event);

        tx.commit();

    } catch (HibernateException e) {

        handleException(e);

    } finally {

        HibernateFactory.close(session);

    }

}

/**
 * Find an Event by its primary key.
 * @param id
 * @return
 */
```

```
public Event find(Long id) throws DataAccessLayerException {
    Event event = null;
    try {
        startOperation();
        event = (Event) session.load(Event.class, id);
        tx.commit();
    } catch (HibernateException e) {
        handleException(e);
    } finally {
        HibernateFactory.close(session);
    }
    return event;
}

/**
 * Updates the state of a detached Event.
 *
 * @param event
 */
public void update(Event event) throws DataAccessLayerException {
```

```
        try {
            startOperation();
            session.update(event);
            tx.commit();
        } catch (HibernateException e) {
            handleException(e);
        } finally {
            HibernateFactory.close(session);
        }
    }

    /**
     * Finds all Events in the database.
     * @return
     */
    public List findAll() throws DataAccessLayerException{
        List events = null;
        try {
            startOperation();
            Query query = session.createQuery("from Event");
```

```
            events =  query.list();
            tx.commit();
        } catch (HibernateException e) {
            handleException(e);
        } finally {
            HibernateFactory.close(session);
        }
        return events;
    }
    private void handleException(HibernateException e) throws
DataAccessLayerException {
        HibernateFactory.rollback(tx);
        throw new DataAccessLayerException(e);
    }
    private void startOperation() throws HibernateException {
        session = HibernateFactory.openSession();
        tx = session.beginTransaction();
```

```
        }
}
```